METHOD OF REALISTICALLY DISPLAYING AND INTERACTING WITH ELECTRONIC FILES

FIELD OF THE INVENTION

The invention relates generally to user interfaces, and in particular to a method of realistically displaying and interacting with electronic files.

BACKGROUND OF THE INVENTION

The desktop analogue for computer user interfaces has been has been popular since the mid 1980s. Both Microsoft Windows (TM) and Macintosh (TM) operating systems have successfully used this user interface (UI).

Over time computer users have adapted to the demands of the interfaces, which have become more powerful over time, but have also increased significantly in complexity. As the complexity has increased, the user interfaces have taken on their own look and feel that at times simulates the desktop, but at others follows their own format paradigm.

As a consequence, users have developed loyalties to particular user interfaces and are consequently very resistant to changes, even with the promise of greater functionality. This is particularly true in rapid turn over applications such as e-mail. With e-mail users have generally had to adapt their style of communications to suit the application being used. While the physicality of buttons and icons have evolved to a more ore less de facto standard, an individuals interaction with the particular user interface is still dictated by that interface to a large degree.

While sophisticated users have been able to evolve with these interfaces, those users who have only recently become exposed to these user interfaces may find significant challenges in relating easily to what is being displayed on screen. As a consequence, there is still a need to provide a more natural or realistic user interface.

SUMMARY OF THE INVENTION

An object of the present invention is to provide an improved method of displaying and interacting with electronic files. Accordingly, aspects of the present invention may include one or more of the following:

- Active Display Signature Authentication;

- Multi Window Natural Documentation Presentation;
- Realistic Page Based Documents;
- Smart Frames;
- Non-rectangular Frames; and
- Programmatic Documents.

The embodiments of the invention have arisen out of and are related to processing of electronic mail (email). However, aspects of the invention set out herein are not restricted to email applications, and can be adapted to electronic documents.

Advantageously, embodiments of the invention set out herein can be used to take an email text message and convert it into a virtual paper message in a multi-window format, including envelope information, business cards, photographs and other such information as may be desired. Embodiments of the invention herein enrich the email environment making it more interesting and much closer to what people are used to seeing in written and printed communications in the real world.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be further understood from the following detailed description with reference to the drawings in which:

Fig. 1 illustrates in a dynamic diagram launching a single frame graphical display application in accordance with an embodiment of the present invention;

Fig. 2 illustrates in a block diagram the relationship of a frame manager to the graphical display application and the frames in accordance with an embodiment of the present invention;

Fig. 3 illustrates in a dynamic diagram, a user launching a graphical display application with a single frame is created and presented to that user in accordance with an embodiment of the present invention;

Fig. 4 illustrates in a block diagram an exemplary relationship between an application, frame manager and various frame windows, in accordance with an embodiment of the present invention;

Fig. 5 illustrates in a dynamic diagram the management of a single instance frame, in accordance with an embodiment of the present invention;

Fig. 6 illustrates a dependency diagram showing the Extension Manager's place in a graphical display application with two frames launched, in accordance with an embodiment of the present invention;

Fig. 7 illustrates in a dynamic diagram how the Extension Manager is ultimately responsible for all the Extension objects, in accordance with an embodiment of the present invention;

Figs. 8 and 9 illustrate how the Extension Manager should first check for the resource set for the given frame within the Resource Provider cache, in accordance with an embodiment of the present invention;

Fig. 10 illustrates in a block diagram the extension manager's handshaking mechanism, in accordance with an embodiment of the present invention;

Fig. 11 a dynamic diagram illustrates the command routing procedure, in accordance with an embodiment of the present invention; and

Fig. 12 illustrates the relationships between the user interface objects, in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The problem is to provide an active display signature that a user would recognize as coming from an individual. The active display signature is to be usable and accessible to the common user. It is to be easy to understand so that a degree of trust can be built up. It is not simply a digital signature but a metaphor for a signature that is easy to recognize and understand.

Embodiments of the invention utilize a bitmap of the actual written signature of the sender, however the bitmap is not used for authenticity purposes. What is desired is a programmatic interface that cannot be spoofed by content. The software automatically deciphers the symbol that is sent and determines if it is authentic. A glyph or an indicator is presented in such a way that it cannot be spoofed by content. As an example, a ribbon or seal can be placed on the document if the signature is authentic. If it is not, the seal with an X through it or a circle and a bar through it can be used. The combination of an embedded signature glyph and active display iconography is used.

Embodiments of the present invention evolve the computer desktop to a multi-window format as opposed to a single document in a window. A multi window format is more natural and simulates a physical desktop to a higher degree.

Applications that use more than one window are known. However, in those known applications the document is only in one window. Embodiments of the invention present a document in multiple windows. For embodiments that are e-mail applications this leads to a "what you see is what was sent" paradigm. The present method results in the desktop being document organized instead of application organized. Consequently, the document runs the show, while the application is pulled in as necessary to act upon that portion of the document that is being worked on. The document is presented in multiple windows each window containing its own information flow organized into one document. In Unix, portable bitmap tools can be used so that one can scale, resize and use contrast and other methods.

This permits a form of piping to take place. A document is created and put through a pipe where it is filtered so it comes out looking very different then the text message that went in. The key is to know the file formats so one can modify it. All applications need to have the particular file format.

- reverse OLE
- imbed application

The embodiments of the present invention provide the windowing functionality as Microsoft Windows (TM) does not support it. A windowing environment is built within Microsoft Windows (TM) in order to display sub-elements of a document in separate windows.

In accordance with an embodiment of the present invention, the metaphor for presentation of document information to use paging rather than scrolling. This supports the idea to use a natural paper look to get the look and feel of a real document. A paper texturizing technique is invented which used a special proprietary compression method. Showing paper texture, including logos, etc., is byte heavy or utilizes a great deal of storage or transmission capacity. The compressive technology set out in a co-pending application U.S. patent application Serial Number 09/262,056, which is hereby incorporated by reference, allows compression to a thousand times smaller than Joint Photographic Experts Group (JPEG) file. It is also resolution independent whereas JPEG becomes very granular as one sizes up. JPEG can also do tiling but it is not very effective. This invention allows the production of virtual paper utilizing compressed textural data. For example, a typical Microsoft Power Point (TM) presentation is a huge information load, with 60 to 70% of that information being with respect to the

background of the presentation. Typically, 50 to 70 megabytes of data gets compressed to approximately 5 megabytes in this application. A key is to be able to put different textures on virtual paper in other applications.

The various embodiments of the present invention are set out in more detail below.

Desired features of the graphical display software architecture (GDSA) are described below:

- Extensible: Products created using this architecture should be as extendible as conveniently and flexibly as possible.
- Convenient: Providing deep extensibility introduces levels of complexity that would normally be non-issues, so creation of products using this architecture should be as convenient as possible.
- Timely: After an initially expensive design phase for the architecture, future products built with the architecture should be able to be produced in a timely fashion.
- Simple: The architecture should be reasonably uncomplicated.
- Clean: Compromises are to be avoided when designing the architecture. The above requirements should be met with little or no shortcuts.

This section describes the general overview of the GDSA. This overview focuses on two major points: The Framework and the Extensibility of that Framework. The Framework is the foundation for the components that implement the core functionality of GDSA products. The complete extensibility of that Framework is the GDSA design goal.

- About the Framework: This section gives a brief synopsis of the GDSA Framework and identifies the major components.
- Initializing the Framework: Once the major components have been identified, their connectivity and interoperability are described. This section illustrates the overall relationship between the major components.
- Extending the Framework: This section gives a general overview on how to add feature functionality to the Frame Windows by using Extensions.

GDSA software is developed by building a basic application, and adding components to it that implement the functionality of that application. Those components are extensible and flexible, so as to be reused in future products. The Framework constitutes the portions of a GDSA Application that are not covered by any one particular

module. There are five general categories of components that constitute the Architecture Framework:

- Applications
- Frame Windows
- Documents
- Extensions
- Services

The Framework can be described as the binding of these components into a coherent system. The following section provides a general overview of the framework and describes how the major components are related and linked together.

Applications

A GDSA application (sometimes referred to as a graphical display application, *infra*) is a small executable file that contains almost no functionality. Some of its basic duties include displaying splash screens (open and close if required) and checking for a previous instance of itself and activating it if a second instance is opened (i.e., Single Instance Application). Unlike a traditional (Microsoft Foundation Class) MFC Application, which consists of the application instance, a main frame window and one or more document-view windows, a GDSA Application is comprised of an application instance and multiple Frame windows (not to be confused with the CFrameWnd in MFC). These frames encapsulate all of the UI that the end user recognizes as the complete GDSA application.

Frame Windows

Frame windows are windows that look and behave like MFC CFrameWnd windows or standalone applications. They have a resizable frame, a title bar, a menu, a toolbar and a main view. The UI may be such that the existence of these parts is not clearly visible, but the functionality is there. A GDSA application may require the creation, usage and management of several frame windows. To the end user, these frames represent the complete application.

In order for anyone create these frames, we require an object that is responsible for creating and managing all frames in the system. This object is the Frame Manager.

The Frame Manager is initially created by the Main Application and is a singleton object accessible to all. The Frame Manager plays one of the most crucial roles in the system and is further discussed in the Initializing the Framework section.

Documents

At the core of each frame window is an object that maintains the basic identity of that frame. This object is called a document object (which, when referred to in the context of the frame window, is sometimes known as the document).

Traditionally, the relationship between document and frame window is 1:1 however a frame window has the freedom to manage multiple document extensions if so desired.

Extensions

Each GDSA Application is installed with a suite of extensions. An extension is an add-on to an application that encapsulates a feature or set of functionality that will allow the end user to perform a specific task.

Extensions can add a new feature or modify the behavior of existing features. As such, extensions have user interface elements, such as menus, toolbars, and Options dialogs. Extensions extend frame windows by inserting themselves into the user interface, giving users access to the extension's functionality. Basically, a frame window builds itself by enumerating through all the extensions that are meant to extend it (specified in registry settings) and incorporating the extensions' functionality within itself. In order to do this each frame window creates an extension manager. The extension manager builds the frame window's toolbars and menus and manages the routing of those commands to the appropriate extensions. The extension manager's role is examined in more detail in the Extending the Framework section.

The Extension Manager is able to figure out which extensions extend the document and establish a connection between them. The extension will communicate and deal with the document (Doc) through its provided interfaces. In some cases, the Docs may need to implement custom interfaces for a specific extension. Currently, GDSA extensions have two flavors: Enhancement Extensions and View Extensions. View extensions are more complex extensions that are in charge of the application's user interface (See View Extension section for detail).

To integrate extensions into the user interface, a smart system is designed to make extension creation as easy as possible, with a slight loss of flexibility, when compared to similar architectures in other applications (such as Outlook (TM)). Since most GDSA applications will be implemented using extensions, the ease of creation became very important. The resulting system solves the expandability and version control problems without overburdening the Extension developer.

Services

A service is a utilitarian object or control that does not implement a specific feature, but provides functionality that is required by other services or extensions. The Frame Manager, for instance, provides the basic means for anyone to create a frame window while the Extension Manager provides these frame windows the means to manage extensions.

Initializing the Framework

Now that major components have been identified, this section describes the starting point of how they all come together to form the framework. The purpose of this section is to illustrate the connections and dependencies between major components in order for anyone to design and develop a GDSA application.

Launching the Main Application

Referring to Fig. 1 there is illustrated in a dynamic diagram launching a single frame graphical display application (or GDSA application), in accordance with the present invention. A user 10 launches an executable (1) causing a GDSA application executable 12 to create a Frame Manager 14 singleton object (2) and requests the Frame Manager 14 to create an initial frame window (3). The Frame Manager 14 creates (4) the initial frame window 16.

A GDSA application cannot be compared to the conventional MFC frameworks that most developers are used to. Although there are familiar concepts such as frames and documents and even similar technology, a GDSA application need not be categorized as Single Document Interface (SDI) or Multiple Document Interface (MDI) or some hybrid of the two. Instead, the GDSA Framework Architecture has opted for a more flexible and less constrained approach to applications.

As described above, a GDSA application is a potentially small executable that consists of an application instance and one or more frame windows. To the end user these frames encapsulate all of the application's UI. Although a user may regard one of those frames as the "primary" or "main" frame that he/she deals with the most, technically there is no single frame window that represents the entire application.

The number and type of frames an application needs is entirely up to that specific GDSA application. A GDSA application can be represented by a single frame (perhaps to appear "SDI-ish") or multiple frames. An application may use a frame that is rather simple in nature or extremely complex (e.g., a single frame that manages multiple documents perhaps...sounds "MDI-ish"). Although the creation of such frames is discussed in more detail in the Frame Window Creation section, this approach is flexible and not constrained to labels.

The Frame Manger 14 helps the main application executable 12 manage the creation and destruction of these frame windows. The Frame Manager 14 singleton Component Object Model (COM) object is a service in a GDSA designed system and is one of the very first objects that a GDSA application creates at startup.

Frame Manager Overview

The Frame Manager is a singleton object initially created by a GDSA application instance. Referring to Fig. 2 there is illustrated in a block diagram the relationship of the Frame Manager to the Application and the frames, in accordance with an embodiment of the GSDA. The Application 12 instantiates the Frame Manager 14, which in turn instantiates the frame windows 16 used by the application.

All creation and destruction of frame windows 16 is done through the Frame Manager object 14. Some of its major functional requirements include:

- Instantiating Frame Windows.
- Keeping track of Frame window creation and destruction.
- Shutdown main application executable when all frames are closed.
- Keeping track of Frame Windows with unique identifiers so as to re-activate them should a request be made to open a similar frame with the same identifier.

This allows GDSA applications to treat their frames as "single instance" (i.e., like trying to open a Word (TM) document twice from within MS Word (TM) or the same message twice from within Outlook (TM)).

The life cycle of a frame window is far from trivial since a GDSA application may allow the user tremendous freedom when it comes to managing the various frames. In the simplest case a user launches the application which presents the user with an initial Frame window. However, an extreme use case scenario may involve the launching a GDSA application, being presented with multiple initial frame windows, and allowing the user to spawn several other frame windows through interaction with the initial frames. As a result the Frame Manager must be prepared to handle these and other equally complicated scenarios.

Frame Window Creation

One of the main functional requirements of the Frame Manager is to create Frame windows. A Frame Window is a COM object, which any client can create through the singleton Frame Manager object. There are potentially several different types of Frame objects in the system, all of which can make up one or more applications. The Frame Window objects by themselves are empty shells, which supply the resizable windows frame, menu bar and toolbar. The substance of the frame window is added later and discussed in the Extending the Framework section. The following sections describe the various Frame scenarios that could be encountered in GDSA applications.

Single Frame Window Creation

The simplest example of an application scenario involves the single Frame Window. Referring to Fig. 3, there is illustrated in a dynamic diagram, a user launching a GDSA application and a single frame is created and presented to that user. The Frame Manager 14 keeps track of all the frame window objects an application has open (in this case, the single one) and is notified if that frame window 16 is closed by the user 10. The Frame Manager 14 manages the frames it creates by maintaining them within an internal list. The notification system between the Frames 16and the Frame Manager 14 are handled by a simple advise-sink mechanism.

An important requirement of the Frame Manager 14can be seen in step (10) of the Fig. 3. Since no single frame window 14 represents the main application, the Frame

Manager's shutdown notification message needs to be sent to a physical window somewhere in order for the application instance to close. Thus, a GDSA application must maintain an invisible window and pass that window handle to the Frame Manager upon the Frame Manager's initialization.

Multiple Frame Window Creation

The Frame Manager 14 can handle a GDSA application 12 that employs multiple frame windows in its system much in the same way as a single-framed application. The difference is only in the number of frames the Frame Manager 14 must manage. As a singleton object in the system, the Frame Manager 14 can receive requests to launch frames from the application instance 12 itself or any other objects created within that applications process. For example, a GDSA application may launch an initial frame 16 through which an end user can launch additional flavours of frame windows. The Polaris (TM) application is such an example.

Referring to Fig. 4 there is illustrated in a block diagram an exemplary relationship between application, frame manager and various frame windows. The Polaris (TM) application 20 instantiates the Frame Manager 14 that instantiates the primary frame window 16. In Polaris, a user is presented with the Primary Frame window 16, through which the user can select and open multiple messages. At this stage, a request is made to the Frame Manager (through its interface) to launch the frames associated with these messages. For example a message editor frame 18, a contact editor frame 20 and a stationary editor frame 22. The Frame Manager 14 creates and maintains a watch over the life cycles of those frames in the same way as in the single frame scenario discussed in the previous section.

Regardless of how many objects make a frame creation request, all frame windows in the GDSA are the responsibility of the Frame Manager 14. It is solely responsible for the creation and destruction of Frame Windows 16-22 and this dependency is shown Fig. 4 for the Polaris (TM) application case.

Single Instance Frames

Referring to Fig. 5 there is illustrated in a dynamic diagram the management of a single instance frame. For more technical information on how to create a single instance frame and the format of the unique identifiers, see the Frame Manager overview section.

In Fig. 5 a user 10 opens a specific message by interacting with the primary frame 16 and as a result a single instance frame 30 is opened at (4). If the user 10 subsequently opens the same message, the Frame Manager at (8) and (9) present the user with the single instance frame 30.

On occasion a GDSA application requires the functionality of re-activating a frame should a request be made by the user to open a new frame window with the same identifier. This behaviour can be seen in applications like Outlook (TM) and Polaris (TM) where a user cannot open the same mail message twice, or within the MDI confines of Word (TM) where an open document is activated if the user tries to open it again. Although some GDSA applications will allow a single document to be opened multiple times, some may chose to follow this "single instance" behaviour.

As a result one of the functional requirements of the Frame Manager 14 is to allow an application instance or any object to create a "single instance" of a frame window based on some unique identifier.

Extending the Framework

Frame windows alone do not provide the necessary substance to make a GDSA application appear complete to the end user. The Frame windows themselves can be seen as empty UI shells that are to be filled or "extended". The following sections describe the role of Extensions in the GDSA Architecture.

Extensions extend Frame windows on two levels. At a core level, an extension can provide a Frame with its basic identity by encapsulating the management and persistence of that frame's data in a single object. To the frame that object represents the document (similar to MFC notion of a document object) and is consequently known as the Document Extension. At a secondary level, an extension can provide a frame with autonomous functionality, such as providing a user with a command to browse their hard drive or view the system registry. More often such an extension is used to extend and enhance the frame's document by perform functions on that encapsulated data (e.g., a frame window that encapsulates a rich text document extension is enhanced by a Format extension that allows the user to change font information). These extensions are known as Enhancement Extensions.

Extensions extend Frame Windows by inserting themselves into the user interface (toolbars /menus/ accelerators), giving users access to the extension's functionality. A

Frame window builds itself by enumerating through all the extensions that are meant to extend it (specified in registry settings) and incorporating the extensions' functionality within itself. In order to do this each Frame Window creates and maintains a watch over the life cycles of those frames in the same way as in the single frame scenario discussed in the previous section.

Extension Manager

The Extension Manager builds the Frame Window's toolbars and menus and manages the routing of those commands to the appropriate extensions. The following sections illustrate this extension construction and their role in the framework in greater detail.

Referring to Fig. 6 there is illustrated a dependency diagram showing the Extension Manager's place in a GDSA application with two frames launched. The GDSA application 10 instantiates and controls the Frame Manager 14. The Frame Manager controls two frame windows 40 and 42 with extension managers 50 and 52, respectively. Extension Manager 50 has documents extensions 60 and enhancement extensions 62. Extension Manager 52 has documents extensions 64 and enhancement extensions 66.

Extension Manager provides the following functions:

- Extension Creation: Describes the steps taken by the Extension Manager to create extensions.
- Integrating Extensions within the Frame UI: Describes the steps taken by the Extension Manager in integrating all of the extensions' UI elements within that of the Frame.
- Document Handshaking: This section describes how the Extension Manager connects the Enhancement Extensions to the Document Extension. It also touches on cases where a Frame may have more than one Doc.
- Extension Command Routing: Describes how the Extension Manager routes specific commands to the appropriate extensions.
- Documents
- Extending the Document with Enhancement Extensions

The Extension Manager is a COM object that provides the Frame with the service of constructing and integrating all of its extensions. There is one Extension Manager per Frame. In the next few sections we will outline some of the Extension Manager's main responsibilities.

Extension Creation

In order for the Frame's Extension Manager to create the appropriate extensions, it first finds a list of those extensions associated with the particular Frame. The location of this list is under a GDSA key in the system registry. The Extension Manager uses the class identifier (CLSID) of the owner Frame to locate that Frame's extension information under the GDSA key in the registry (the interface details are discussed in technical detail in the Extension Manager reference section). For example, the registry may appear as follows:

[...]\
    \Frames
        \CLSID of Frame
            \[Extension Info]
            \ CLSID of Document Extension
            \ CLSID of Enhancement Extension 1
            \ CLSID of Enhancement Extension 2
            \...

Once the list is located, the Extension Manager 50 enumerates through and instantiates the Extensions 60 that are extending the Frame 40. The Extension Manager 50 is ultimately responsible for all the Extension objects it creates and is responsible for handling their destruction. This is illustrated in Fig. 7 the dynamic diagram.

Integrating Extensions within the Frame UI

Once the Frame's extensions are created, the Extension Manager 50 proceeds with the integration of the extensions within the Frame's user interface. In order for an extension to extend the Frame 40 it supplies the end user 10 with access to its features through user interface components. These components include the menu, toolbar and accelerators (although there is no visible UI, an accelerator key still provides access to the extension's feature).

Given the Frame's menu/toolbar/accelerator resources, again through an Extension Manager initialization interface, the Extension Manager 50 begins to merge them with the resources of each extension in priority order: Frame, Document Extension, and then Enhancement Extensions. This request is made of the Extensions through the standard Extension interface. The rules governing this resource merge, including conflict resolution of command IDs, is described in detail in the Structured Resource Insertion section. The final set of resources is handed back to the Frame for use within its UI.

When dealing with GDSA applications that allow the user to open numerous instances of a particular Frame object (which happens quite frequently in Polaris (TM) whereby a user is allowed to open several messages at once), we must find some means of optimizing the Extension Manager's resource merging. The Extension Manager constructs a Frame's resources based on the class of Frame rather than the instance of a Frame. Each class of Frame is extended by a set of extensions, a set listed in the registry. Thus, if an Extension Manager is constructing the resources of a particular Frame instance it should cache those resources so another similar Frame/Extension Manager pair can use it. The location of this cache can logically fall to the Resource Provider. The Resource Provider supplies an interface so that anyone in the system can use it as a resource cache.

As shown in the dynamic diagrams of Figs. 8 and 9, an Extension Manager 50 should first check for the resource set for the given frame within the Resource Provider cache 70. If there is none to be found, the Extension Manager proceeds to create the merged resources and upon completion, sets the resource set into the cache. Should another similar Frame come along, its Extension Manager can retrieve the cached resources and save itself the manual work of recreating them.

Handshaking

Referring to Fig. 10 there is illustrated in a block diagram the extension manager's handshaking mechanism. In order for an Enhancement Extension 62 to extend the Frame's document, the Extension Manager 50 must provide a handshaking mechanism between all enhancements and their document. Once this handshaking is complete, the Enhancement Extensions 62 can query specific interfaces on the Document Extension 60 in order to interact with the document data. This interaction is described in further detail in the Extending the Document with Enhancement Extensions section.

The handshaking mechanism begins with the Extension Manager 50 instantiating the Document of a Frame and then proceeds to the creation of the Enhancement Extensions 62. As it creates each enhancement, it provides them with an interface pointer to the Document, which they hold on to for later use.

Extension Command Routing

When the process of merging the resources begins, the Extension Manager 50 handles potential conflicts between extensions and the Frame's own commands (such as File-Close or Help). In order to avoid messy conflicts, the Extension Manager 50 re-maps all of the extension commands into a command map of its own.

As the Extension Manager 50 loops through each of the extensions' resources it follows these steps for each extension command encountered:

- Given an extension command ID, store the extension pointer and original command ID within an internal lookup table.
- Assign that extension command a new and unique ID.
- Insert that new command into the final resource set.

Thus, when the user invokes that command from within the Frame 40 through the menu, toolbar or accelerator, the command is routed through the Extension Manager 50. The command is then translated through the Extension Manager's internal map and forwards the original command ID to the extension.

This command routing procedure allows the Frame 40 to simply keep its own message map and IDs since the Extension Manager 50 only translates extension commands. Any command that is not caught by the Frame's message map falls through to the Extension Manager's map. In Fig. 11 a dynamic diagram illustrates the command routing procedure.

Documents

As previously mentioned the Document or Doc represents the core of the Frame. The Doc provides the majority of the functionality perceived by the user. Since a Doc is an encapsulation of a data type, it provides the basic means to manipulate that data. When designing a Doc one categorizes all basic functionality within it and design Enhancement Extensions to handle any special data handling. To facilitate the special

handling, a handshaking mechanism (described in the earlier section) is utilized to allow Enhancement Extensions access to the document data. This handshaking mechanism gets complicated when dealing with multiple documents per Frame, an allowable scenario in the GDSA architecture.

Usually the relationship between Frame and Document is 1:1. However, on occasion a Frame may actually manage more than one Document Extension at a time (e.g., Polaris' Primary Frame manages 3 documents). This management of multiple documents is left largely up to the Frame itself, however it does use some functionality provided by the Extension Manager 50. The first sign of help in managing multiple documents can be found within the registry (See registry hierarchy for an accurate application view of the structure):

[Metamail]\
    \Frames
        \Initial Frame
            \[Extension Info]
                \Doc1\[Enhancement Ext. List]
                \Doc2\[Enhancement Ext. List]

where the Frame16 managing multiple documents has its enhancement extension list broken up by document. This allows the Extension Manager 50 to handshake the Enhancement Extensions 90 and 100 it creates with their associated documents 92 and 102. The second sign of help comes in the form of the Extension Manager 50 returning separate merged resources for each document. This is done through the Extension Manager's interface whereby it returns an enumeration of resources sets, one per document. What the Frame 16 does with the resources (i.e., how it switches between them, etc.) is the Frame designer's choice. The relationships are shown in Fig. 12.

Extending the Document with Enhancement Extensions

Once the document handshaking has been completed between the Extensions and Document, the Extensions need to access the document data through specific interfaces. These interfaces are defined by the Document designer with the intent of exposing as much data as possible (or is allowable) to other extensions. A Doc designer gathers such information by examining existing extensions and anticipating future ones.

For example, a Text Doc may expose basic interfaces to retrieve selected text or the complete text stream from the document and allow modifications on that text. The designer may have used the needs of an existing Enhancement Extension, such as the Spell Checker, to determine what interfaces to expose from the Doc. With this basic set of interfaces established, a future Enhancement Extension, that gets the selected or complete text and converts it all into French, can be written by a 3rd party developer. Similarly, an Image Doc may give Enhancement Extensions access to the mask drawn by the end user so that future paint effects extensions can be written.

Window Frame Management

Embodiments of the invention which involve the management of multiple frame windows for a single document include the following details.

In keeping with the concept that the many frame windows present various portions of the single document in a realistic fashion, embodiments of the invention were designed specifically to handle the application of standard user interactions on this uniquely presented user interface.

This includes but is not limited to:

- Coordinated Resizing
- Coordinated Moving/Layering
- Coordinated Minimizing
- Coordinated Restoring/Maximizing/Task Switching

Coordinated Resizing

Consider an email document with two frames. One contains the envelope, the other contains the letter proper. If the user resizes the envelope, in this case, causing a zoom-effect, since the contents of the envelope can be made to stretch to fit the available frame window dimensions, then the main letter proper should also resize appropriately, in order to maintain a proper visual cohesion and realism.

Embodiments of the invention, specifically the Frame Manager and Frame Windows, coordinate the various Operating System and User Interface messages and drive the resizing of all interdependent frames through complex sequences of code.

Essentially, as the frame window is resized by the user, the Frame Window notifies the Frame Manager, which then sets all other dependent frame windows to a "automated resize" state, then resizes those windows according to their dependency rules. Based on their state, each frame window intelligently determines where to send any corresponding notifications. So, for instance, frames in the "automated resize" state will not notify the Frame Manager when they detect their size changing. Thus the Frame Manager will not become confused by receiving size notifications that were caused by the implementation of the coordinated resizing.

Frame window size dependency rules allow complex interactions to be defined. For instance, all frame windows can be locked, so the user can perform no resizing. Or zero, one, or more frames can be flagged for resizing by the user, while zero, one or more frames can be "anchored" to those user-movable frames, or any other frame. Further, specific borders of any given frame can be likewise flagged for movement or anchored to other sides of other frames. Thus, in the example above, the user might only be able to resize the right and bottom sides of the main letter frame, and only resize the left and top sides of the envelope frame.

Coordinated Moving/Layering

Consider an email document with two frames. One contains the envelope, the other contains the letter proper. It may be that if the user moves the envelope, then the main letter proper should also resize appropriately, in order to maintain a proper visual cohesion and realism. Apropos, a frame may also be moved independently, without affecting the other frames. Consider if a business card was included in the same example. The user may wish to drag the business card to a different location relative to the main letter proper.

Embodiments of the invention implement such complex movement rules and flexibility. Further, restrictions can be applied on the movement examples described above. For instance, the business card can be tagged as being "attached" to the main letter frame. In such a case, if the user tries to drag the business card completely off the paper, the Coordinate Moving system would allow the user to freely move the business card over the page, but it would prevent the user from moving the business card in such a manner that it is completely off the reference page.

Further, different regions of the frames can be used to initiate the movement command. For example, the main menu frame has a title bar, and, as is standard in Microsoft Windows related operating systems, when the user clicks and drags the mouse on the title bar, the window moves accordingly. In the example above, the menu frame resides above the envelope, and moving the frame using the title bar moves all frame windows in the document. However, the business card does not have a title bar of its own, and thus allows for user movement by accepting drag points from any location on the business card. Unlike the menu frame movement, the business card can be tagged in such a fashion that dragging the business card will only move the business card, while leaving all of the other document frame windows be.

Essentially, the Frame Windows set up their movement regions, then handle movement commands delivered by the Operating System. According to their movement rule tags, the Frame Windows will notify the Frame Manager of the movement, and the Frame Manager will determine how to handle the movement. Conversely, the Frame Window might also handle some of this logic, bypassing the overhead of the Frame Manager if it is not required. If the movement flags indicate that other windows should move as well, the Frame Manager is used to set the state of all of the other windows to an "automated move" state, and then moves them accordingly. Frame Windows that are in the "automated move" state do not notify the Frame Manager of their movement. This prevents the frame manager from confusing movement invoked by the Coordinated Movement algorithms from movement invoked by the user. Other techniques to implement this process could also involve allowing the Frame Manager to handle all movement and to implement all business logic rules for the coordinated movement, removing the filtering of messages from the Frame Window logic. In this case, Frame Windows would always pass all movement notifications to the Frame Manager which would ignore those that were not invoked by the user – here the Frame Manager must track the user movement separately from the coordinated movement logic.

Layering uses a similar mechanism, save that it handles the movement of the frame window's z-order on the operating system's desktop. The "z-order" determines which frame window appears in front of, or on top of, the other frame windows. A window that is behind another window will be obscured by a higher window (in front or on top).

Typically, the user controls the z-order of frame windows simply by activating them (clicking on them or switching to them using keyboard commands or other mechanisms). The Windows Operating System automatically brings the activated window to the top of the z-order so the user can see the contents of that window.

Embodiments of the invention allow for flags to be set within the frames that control the frame windows location with respect to all of the other frame windows in the document. Some frame windows, for instance, can be always kept behind, in front of, or between other frame windows. Other frame windows can cause notifications to be delivered through the Frame Manager and Extension system to allow functionality to be invoked when the user attempts to bring them to the foreground by activating them – thus allowing for bookmarks and other such UI elements, which automatically reconfigure the frame layout.

Essentially, the Frame Windows intercept notifications from the Operating System when the z-order of the window is to change, and either determines to reject the change, or notifies the Frame Manager to allow it to determine how to handle the z-order change. In either case, once again, the other frames may need to be updated, and the Frame Manager is once again responsible for setting the frames to an appropriate state, and chaining further z-order changes through the frame windows as indicated by the layering rule tags.

Coordinated Minimizing

Embodiments of the invention also cover coordination of user requests for minimizing the document or switching between the document and other running applications on the Operating System, and back again.

When the user requests a minimization of the document, the Frame Manager minimizes the main encapsulating frame window, and presents only a single application on the operating system's task bar. This is handled through a sequence of calls to hide all of the frame windows of the document.

Coordinated Restoring/Maximizing/Task Switching

Embodiments of the invention also cover coordination of user requests for restoring, and maximizing the document. These are very similar to the process described in the Coordinated Moving and Coordinated Resizing embodiments.

Maximization requests are handled exactly like resize requests, save that the maximized size is determined automatically according to the size of the desktop and the layout of the frame windows. The frame windows are resized in such a fashion that they are all visible and that they fit the desktop to the maximum size possible without spilling beyond the edges of the display. This is implemented through simple mathematical determination of the relative sizes of the frames, their locations, and the desktop window.

Restoring shows the frame windows and moves them to the location they were in prior to being minimized. This is implemented by chaining commands sequentially through all of the frame windows using the mechanisms described in the Frame Manager sections above.

When the user requests to switch from the document to another application running on the operating system, and then back again, the Frame Manager represents all frames to the Operating System as a single application, so that only a single entry appears in the "alt-tab" list or in the start menu bar for this document. When the user choose the document to switch back to it, the Frame Manager treats this as a restore (see above).

Realistic Page Based Documents

Embodiments of the invention which involve the management of multiple frame windows for a single document include the following details.

In keeping with the concept that the many frame windows present various portions of the single document in a realistic, page-based fashion, embodiments of the invention were designed specifically to handle the application of standard user interactions on this uniquely presented user-interface.

This includes but is not limited to:

- Encapsulating a Single Page
- Browsing Pages (Bookmarks and Page Curls)

Encapsulating a Single Page

Embodiments of the invention were designed to allow for visual representation of an entire "page" for the purposes of complete viewing of the document without the requirement for manipulation of scroll bars, or any other mechanism that requires the user to move a virtual window around on a document to see all of the details of a single page.

All visible content is broken up into pages, where each page represents the amount of information that can be displayed within the screen area allocated to that content element. For instance, the main letter of an email message is split into pages by the size of the window used to contain the text. A yellow sticky-note on the side is broken down into many more pages to account for the restricted screen real-estate of the sticky-note window.

In this fashion, the user always sees a complete representation of one page of data when they view the document, without the need to use scrollbars, cursor keys, mouse wheels or other mechanisms to move a virtual "view-port" over the single page of the document.

Browsing Pages

A single page curl (or equivalent keyboard or mouse input target) can be used to "flip" from the current page to the next. A second page curl can be used to flip to the previous page, when previous pages are available. All content, including frame windows and z-order of frame windows, can change according to the page number. This allows, for example, business cards and photographs to be attached to specific pages in the document. The keyboard can also be used to page forward and backward.

A single frame can be tagged in such a fashion that paging that frame does not page the entire document. In the aforementioned sticky-note example, a multi-page sticky-note can be attached to the first page of a letter. Paging through the sticky-note would not cause the main letter page to change. However, paging through the main letter would cause the sticky note to change z-order, considering that the page that the sticky-note was attached to is no longer visible.

This is implemented by communicating paging commands through the Extension system. This eventually winds up at the document as a page change notification. The document then notifies all interested parties – the Page Item Controls, the Extension Manager, the Frame Manager, etc., which causes a cascade of events – essentially, the new user-interface description for the next page is loaded, all user-interface controls are created for all visible frames for that page, and the controls are loaded with the appropriate page's contents. Some frame windows may also be resized, moved, hidden or z-ordered (see the Window Frame Management section for details).

Smart Frames

Aspects of the invention which involve dynamically determining content to display are encapsulated in the Smart Frames embodiment.

Multiple streams of content can be delivered in the document for any given Smart Frame or element on a Frame (Page Item). The envelope around these multiple streams includes an algorithm to run to determine which stream of data to use to populate the Smart Frame. The Smart Frame reads the algorithm, retrieves data from local sources, applies that local data to the algorithm, and determines which stream of content to display based on the results.

In this way, the Smart Frames system allows content providers to target specific categories of users based on the data the Smart Frame can access. This data can include, but is not limited to, such things as: computer hardware configuration, installed software configuration and version information, demographic information (as provided by the user), preferences (as provided by the user), etc.

For example, a company that sells various audio/video devices for PC's might use the Smart Frame technology to deliver advertisements for their latest video card, sound card and speakers. The algorithm delivered in the Smart Frame stream would request the version of the user's video card, sound card and speakers, and display first the video card if it was newer than the users, then the sound card if it was newer than the users, then the speakers if they were newer than the users, then a generic ad for the company if none of the previous three conditionals were met.

For example, imagine if the user had an older video card. The Smart Frame would show the video card advertisement the first time the user opened the document. After purchasing a new video card, if the user were to open the same document, the Smart Frame would show the sound card advertisement, since the first conditional in the algorithm would no longer be met.

Non-Rectangular Frames

Aspects of the invention which involve displaying non-rectangular frame windows are covered in this embodiment.

The Frame Windows described in the Frame Manager sections and the Window Frame Management section in this addendum can be tagged in such a way that the rendering system will display them using an arbitrarily shaped border.

This border can be represented through the use of a masked or alpha-blended bitmap image, which may also be compressed using various image compression techniques. Or this border might also be represented through the use of a vector-based definition, which could involve polygons or Bezier curves or similar vector-based techniques.

Optionally, these non-rectangular frames can be broken down into sections which would algorithmically be used to render the frame. Sections could include, for instance, top/left, top, top/right, left, middle, right, bottom/left, bottom, bottom/right. These sections could then be rendered over any arbitrarily sized Frame by rendering the top/left section at the top/left corner of the Frame Window, repeating the top section as required across the top of the Frame Window, then rendering the top/right section at the top/right corner of the Frame Window. The left section would be repeated as required to cover the left edge of the Frame Window, while the right section would similarly cover the right. The bottom edge of the frame would be rendered in a fashion similar to the top edge, except that the bottom/left, bottom, and bottom/right sections would be used. The middle would repeat across the section of the frame not covered by the edges. In this fashion, the non-rectangular window could be resized to any arbitrary size without resizing the content of the window borders.

Optionally, if resized, the non-rectangular Frame Window would simply scale the border according to the new size. In this case, if the border is represented using a bitmap image masking or alpha technique, then this scaling might also include filtering to minimize bitmap image scaling artefacts. If the border is represented using a vector-based technique, then the vectors would scale cleanly to the new size without concern for aliasing or other artefacts.

Optionally, each edge of this border can be also made resizable.

If the border is defined by a vector-based technique, then this can be implemented using a normal to the tangent described by each location on the border, and projecting that normal into the Frame Window to a distance determined by querying the Operating System for the border width property. The resize cursor (and therefore the behavior of the resize when the user clicks and drags) could be defined by the angle of the normal. Resize cursors include "left edge", "top/left edge", "top edge", "top/right edge", "right edge", "bottom/right edge", "bottom edge" and "bottom/left edge". The Operating

System defines these cursors and describes the behavior to implement when the user clicks and drags over one of those resize areas.

If the border is defined using a bitmapped image based technique, the resizing area could be defined by encoding the border resize areas within the image. Color, bit depth, or some other bit-based property could be used to indicate which resize area the mouse currently lies above.

Alpha-blending can also be applied to definition of the non-rectangular frame and then applied to the Frame Window while rendering. When coupled with the z-ordering features described above, non-rectangular frame windows can overlap and complex rendering effects can be made available by overlapping alpha-rendered windows.

Programmatic Documents

Aspects of the invention which involve integration with local applications, scripted control of Frame Windows, Smart Frames, and Document Content, all fall into this embodiment.

By implementing support of XML-based scripting technologies, document envelopes can contain script to control, for example:

- Movement, Creation, Destruction, Hiding, Showing, Browsing and Populating the Frame Windows and Smart Frames, and the underlying Document itself.
- Integration with local, pre-installed applications
- User-interface interaction
- Security features

Essentially, the idea is to deliver a document which contains all of the functionality of a full application without the need to download and install the application. The invention implements all user-interface and control features, and provides the document script with access to various capabilities, which allow a content provider to essentially deliver an application directly to their customer.